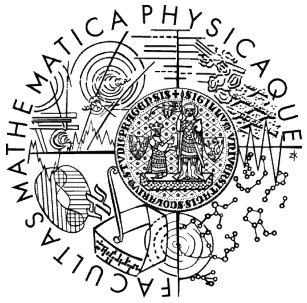
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA



Peter Kmeť

Delta komprese

Kabinet software a výuky informatiky

Vedúci bakalárskej práce: Mgr. Martin Senft
Študijný program: Informatika, Správa počítačových systémů

2007

Na tomto mieste by som sa chcel poďakovať vedúcemu bakalárskej práce Mgr. Martinovi Senftovi za užitočné návrhy a pripomienky, ktoré mi poskytol pri vývoji programu, svojim rodičom za sústavnú podporu, bez ktorej by táto práca nemohla vzniknúť a Márií Kolovratníkovej a Pavlovi Jančíkovi, službám vo fakultnom počítačovom laboratóriu v budove VŠK 17. listopadu, za neoceniteľnú pomoc pri tvorbe jej finálnej podoby.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním svojej práce a jej zverejňovaním.

V Prahe dňa 15.08.2007 Peter Kmeť

Obsah

Názov práce: Delta komprese
Autor: Peter Kmeť
Katedra: Kabinet software a výuky informatiky
Vedúci bakalárskej práce: Mgr. Martin Senft
e-mail vedúceho: Martin.Senft@mff.cuni.cz

Abstrakt: V tejto práci prezentujem algoritmus na riešenie problému delta kompresie spočívajúceho v otázke ako úsporne reprezentovať zmeny, ktoré nastali v upravenej verzií súboru vzhľadom na pôvodný súbor. Algoritmus využíva metódu hľadania v slovníku na nahrádzanie opakujúcich sa reťazcov znakov krátkymi odkazmi na ich predchádzajúci výskyt, pričom sa zhody zisťujú v oboch súboroch. Spoločné predpony reťazcov sa získavajú prostredníctvom sufixových polí, potenciálne zaujímavej alternatívy k iným postupom. Riešenie je rozdelené do niekoľkých ľahko zmeniteľných modulov a rozdiely sa ukladajú v súboroch vo vlastnom formáte. V práci sa stručne diskutujú výsledky použitých techník a navrhujú vylepšenia projektu.

Kľúčové slová: delta kompresia, slovníkové kódovanie, sufixové polia

Title: Delta Compression
Author: Peter Kmeť
Department: Department of Software And Computer Science Education
Supervisor: Mgr. Martin Senft
Supervisor's e-mail address: Martin.Senft@mff.cuni.cz

Abstract: In this work I present an algorithm for the delta compression problem referring to effective representation of changes, which occur in a modified version of the file with respect to the original file. The algorithm uses the method of searching a dictionary to replace repeating strings of characters by short references to their previous occurrence. These matches are located in both files. Common prefixes of strings are obtained using suffix arrays, potentially interesting alternative to other procedures. The solution is divided in couple of easily modifiable modules and changes are stored in own file format. In the work results of used techniques are briefly discussed and possible improvements of the project are proposed.

Keywords: delta compression, dictionary coders, suffix arrays

Kapitola 1

Úvod

1.1 Popis problému

Dramatický nárast veľkosti softvérových projektov a počtu programátorov spoločne pracujúcich na jednom projekte si v posledných rokoch vyžiadal vznik nových postupov umožňujúcich komplikovaný proces vývoja zjednodušiť a koordinovať. Medzi často používané riešenia z tejto oblasti patria aj systémy na správu verzií, ako príklad možno uviesť programy Subversion, Git alebo Darcs. Úlohou takéhoto systému je zaznamenávať všetky zmeny v projekte, aby sa dala v prípade požiadavky efektívne sprístupniť ľubovoľná verzia vyvíjaného softvéru, nazývaná tiež revízia.

To sa dá dosiahnuť odlišnými spôsobmi, na ktorých závisí, ako sa v systéme popisujú úpravy v zdrojových súboroch či dokumentácií. Jednou z možností je priamočiary prístup, pri ktorom sa ukladá kompletná podoba projektu. Pri prechode od jednej revízie k inej sa potom zmenené súbory jednoducho prepisujú. To však vedie k zbytočnému plytvaniu priestorom, v prípade distribuovaných riešení aj kapacitou spojenia, keďže rozdiely medzi za sebou idúcimi verziami obvykle tvoria len malý zlomok celkovej veľkosti súborov.

Z toho dôvodu sa dnes vo väčšine nástrojov udržuje v celom rozsahu len prvá verzia projektu, ďalšie revízie sú ukladané vo forme takzvaných delta súborov. Jedná sa o záznam zmien, ktoré treba v jednom zo súborov vykonať, aby sa zhodoval s druhým súborom. Práve spôsob vytvárania delta súborov je hlavnou témou tejto práce.

V súčasnosti existuje niekoľko programov, ktoré implementujú delta kompresiu. Klasickou technikou je použitie niektorej z variácií softvéru diff [1]

v kombinácií so štandardnou bezstratovou kompresiou. Diff porovnáva dva textové súbory po riadkoch a na výstupe zaznamenáva nájdené rozdiely (novšie verzie pracujú aj s binárnymi súbormi). Následne sa výstup komprimuje, napríklad pomocou štandardného nástroja gzip.

Pred niekoľkými rokmi však Korn a Vo predstavili metódu vdelta [2], v ktorej sa zisťovanie zmien a ich úsporné ukladanie nedeje v dvoch nezávislých krokoch. Naopak, klasická kompresia sa považuje za špeciálny prípad porovnávania súborov, pričom referenčný súbor má nulovú veľkosť. Hlavnou myšlienkou je uplatnenie slovníkového algoritmu v štýle LZ77 [3] s tým, že okno slovníku sa plní dátami z pôvodného aj upraveného súboru. V tomto okne sa hľadajú predchádzajúce výskyty reťazcov v súbore a nahrádzajú sa príslušnými odkazmi. Odkazy zaberajú menej znakov, dosahuje sa tým teda žiadaná redukcia spotrebovaného miesta.

V porovnaní s dvojfázovými spôsobmi vdelta dosiahla lepší čas aj kompresný pomer pri spracovaní rôznych typov dát z projektov GCC a Emacs [4]. Následne sa začali objavovať ďalšie riešenia tohto typu, napríklad xdelta a zdelta, s vylepšenými postupmi na vyhľadávanie zhodných reťazcov, posúvanie slovníku a ukladanie dát. Tým sa ešte viac urýchlila kompresia a zmenšila veľkosť výstupných súborov.

Postupy kompresie samostatných veľkých súborov tiež postupom času napredovali. V snahe obmedziť pamäťovú náročnosť sufixových stromov, využívaných okrem iného pri vyhľadávaní vzorov v DNA sekvenciách, Manber a Myers navrhli štruktúru nazývanú sufixové pole [5]. S ňou pribudla možnosť ako rýchlo identifikovať všetky výskyty nejakého podreťazca v určitom reťazci. Pole pri tom plní úlohu indexu sprístupňujúceho podreťazce v lexikografickom usporiadaní. Vďaka tejto vlastnosti sa skrátil čas potrebný na určenie predošlých výskytov reťazcov pri kompresií s využitím slovníku.

Kým integrácií sufixových polí do algoritmov na kompresiu jednotlivých súborov sa venuje čoraz väčšia pozornosť, ich eventuálnym prínosom pre problematiku delta kompresie sa zaoberá len málo výskumov. Javí sa byť teda prínosné a zaujímavé overiť dopad použitia týchto polí na vytváranie delta súborov a porovnať výsledky s doterajšími postupmi.

Cieľom tohto projektu bolo naimplementovať a popísať nástroj na delta kompresiu pracujúci na princípe slovníkovej metódy s hľadaním opakovaných výskytov reťazcov v sufixovom poli. Súčasťou práce je stručná diskusia o výsledkoch programu, jeho slabých stránkach, predpokladoch na použitie a perspektívnych vylepšeniach.

1.2 Obsah kapitol

V texte práce sa používa *naklonené písmo* pre názvy premenných, polí či indexov, `neproporcionálne písmo` pre názvy knižníc, tried, funkcií a iných odkazov na zdrojové kódy.

Kapitola číslo 2 sa zameriava na bližšie vysvetlenie navrhnutého riešenia. Obsahuje popis jednotlivých častí programu a najdôležitejších konkrétnych tried a metód, ktoré implementujú výsledný algoritmus. Uvádzajú sa v nej aj dôvody, ktoré viedli k voľbe použitých prostriedkov a postupov. Každá podkapitola sa venuje jednému zo šiestich samostatných modulov, do ktorých je celé riešenie rozdelené.

Pre lepšie pochopenie algoritmu môže byť užitočné nahliadnuť do zdrojových súborov modulov, ktoré sa nachádzajú na priloženom disku. Sú komentované tak, aby sa v nich dalo čo najjednoduchšie orientovať. V záhlaví je vždy umiestnená krátka informácia, na čo daný modul slúži. Objektový model je vysvetlený v hlavičkovom súbore, implementácia metód v zdrojov súbore. Pred každou časťou modulu je k dispozícií stručný výklad jej funkcie. Zložitejšie metódy majú komentované vstupné parametre, prípadne aj návratové hodnoty. Na pravom okraji kódu je umiestnená legenda, ktorá dopĺňa ostatné komentáre a približuje komplikovanejšie miesta v algoritmoch.

Kapitola 3 sa zaoberá použitím programu a hodnotí jeho výsledky. Jej súčasťou je užívateľská dokumentácia s pokynmi na zadávanie príkazov a popisom dostupných parametrov. V kapitole tiež možno nájsť výstupy z niekoľkých základných testov zameraných na výkon programu v zmysle dosiahnutého času a kompresného pomeru. Tieto údaje sú porovnávané s parametrami klasického riešenia využívajúceho nástroje diff a gzip. Nakoniec sú stručne diskutované nedostatky predloženého návrhu a naznačené spôsoby ako ich odstrániť.

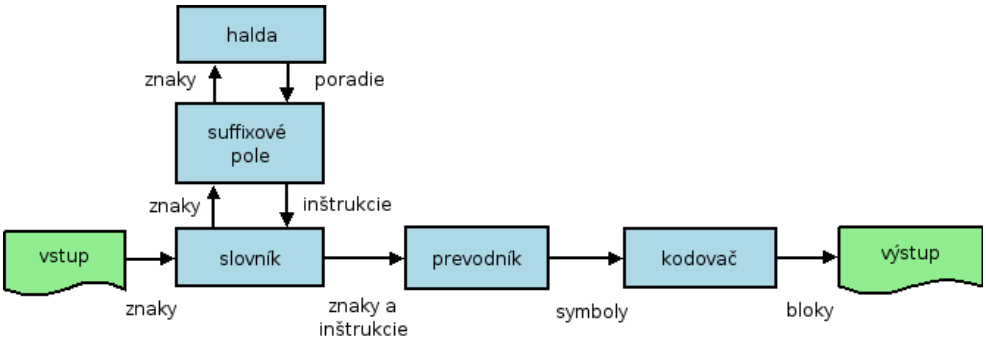
V kapitole 4 sa všeobecne hodnotí celý projekt a v závere sú uvedené odkazy na použitú literatúru.

Kapitola 2

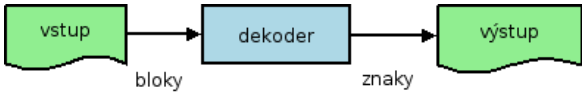
Popis riešenia

Výsledný program je implementovaný v programovacom jazyku C++. Pre potreby tohto návrhu ide o vhodný kompromis medzi rýchlosťou nižších jazykov a komfortom vyšších jazykov. Jedným z dôvodov výberu je aj dobrá podpora objektov a šablón. Tieto nástroje sú využité na dosiahnutie vyššej univerzálnosti a jednoduchšej zmeniteľnosti kódu. Okrem spomenutých kritérií sa bral pri implementácií zreteľ na nevyhnutnú prenositeľnosť medzi operačnými systémami - program sa spolieha len na malý počet štandardných knižníc jazyka.

Riešenie je rozdelené do niekoľkých logických komponentov. Ich využitie v programe a vzájomné prepojenie je znázornené na schémach kompresie a dekompresie, na Obrázkoch 2.1 a 2.2.



Obrázok 2.1: Schéma kompresie



Obrázok 2.2: Schéma dekompresie

Zelenou farbou sú vyznačené vstupné a výstupné súbory, modrou farbou celky zodpovedné za rôzne štádiá vykonávanej operácie. Na vytvorenie predstavy o komunikácií medzi jednotlivými časťami slúžia šípky približne symbolizujúce toky dát a popisy pod nimi udávajúce typy prenášaných dát.

Proces kompresie prebieha takto: zo spracovávaného a porovnávacieho súboru sú postupne načítavané znaky do slovníku. Slovník na základe znakov iniciuje vybudovanie štruktúr haldy a sufixového poľa. Tie používa na lexikografické zoradenie podreťazcov a nasledovnú identifikáciu zhodných predpôn v rámci vlastného obsahu. Podreťazce so spoločnou predponou predstavujú opakované výskyty za sebou idúcich znakov.

Každý opakovaný výskyt v slovníku sa nahrádza spätnou referenciou v tvare [*dĺžka*,*offset*] do predchádzajúcich dát jedného zo súborov. To sa dá interpretovať ako informácia koľko znakov z ktorej predošlej pozície je potrebné nakopírovať na súčasnú pozíciu, aby sa obnovil opakovaný výskyt. V terminológií delta kompresie sa preto podobné odkazy tiež nazývajú kopírovacími inštrukciami.

Postupne vytvárané inštrukcie a pôvodné znaky, pre ktoré sa nepodarila nájsť zhoda sa posúvajú do prevodníka, neformálne označovaného ako „symbolizátor“ (z anglického výrazu pre znázornenie - „symbolization“). V rámci neho sa získané údaje prepisujú do podoby symbolov vo vlastnej abecede, aby sa s nimi dalo v ďalších krokoch lepšie pracovať. Poslednou fázou je ukladanie symbolov do blokov výstupného delta súboru, z ktorých sa dajú neskôr obnoviť pôvodné znaky. Deje sa tak v module kódovača.

Pokiaľ ide o dekompresiu, situácia je o niečo jednoduchšia. Keďže nie je potrebné zisťovať žiadne zhody a pri obnove pôvodných znakov nie je zložité narábať priamo so symbolmi, túto funkciu zabezpečuje jediný objekt. Vstupom je pôvodný porovnávací súbor a delta súbor. Dekódovač číta delta súbor po blokoch a z nich určuje symboly na spracovanie. Podľa charakteru symbolov posiela na výstup buď priamo načítaný blok znakov alebo vykonáva kopírovaciu inštrukciu nasmerovanú do delta súboru respektíve do porovnávacieho súboru, ktorou požadované znaky získava. Všetky znaky nakoniec zapisuje do výstupného súboru. Výsledok vedie k pôvodnému

porovnávaciemu súboru po aplikácií zmien zaznamenaných v delta súbore.

Celková schéma nie je komplikovaná, ale pri návrhu konkrétnych mechanizmov je nutné urobiť niekoľko rozhodnutí s významným dopadom na celé riešenie a vysporiadať sa s výnimočnými situáciami, ktoré môžu nastať. Týmto problémom je venovaný podrobnejší rozbor zvlášť pre každú súčasť programu.

Pred ním je vhodné upozorniť na fakt, že prevažná väčšina tried a metód, ktorými sa text zaoberá, je šablónovaná a očakáva sa pre ne určenie dvoch dátových typov - `elem_t_` a `size_t_`. Prvý udáva typ prvkov, ktoré sa rozoznávajú pri načítavaní údajov, ukladaní do vyrovnávacích pamätí či porovnávaní. Druhý typ slúži na zadávanie veľkostí, napríklad dĺžky slovníka a rôznych vnútorných štruktúr. V hotovom programe sú nimi typ `char` a `unsigned int`.

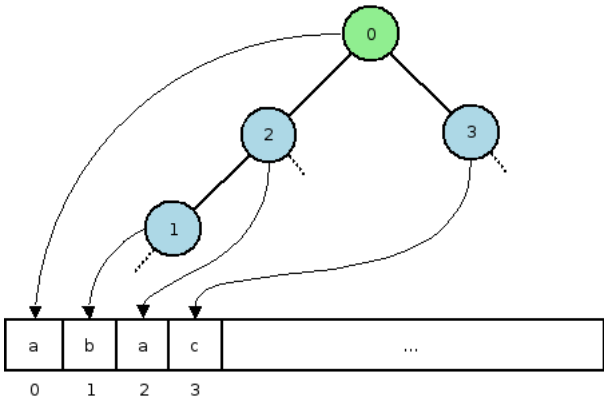
Bohužiaľ sa pri implementácií nedalo vyhnúť menším častiam priamo závislým na konkrétnych typoch. Navrhnutý binárny formát výstupných súborov si vyžaduje čítanie a zapisovanie dát po jednotlivých bytoch, čo môže vyvolať problémy pri typoch prvkov inej veľkosti. V rámci formátu je tiež potrebné stanoviť hranice blokov - počíta sa v nich s maximálne štvorbytovými hodnotami. Je zodpovednosťou programátora, aby pri prevzatí šablón vykonal patričné úpravy, ak je to nutné.

2.1 Minimálna hlada

Na postavenie sufixového poľa pre reťazec zo slovníku je nevyhnutné poznať lexikografické usporiadanie všetkých jeho podreťazcov. Základný prístup ako ho určiť spočíva v aplikácií niektorého efektívneho triediaceho algoritmu. Vo výslednej knižnici padla voľba na vlastnú jednoduchú implementáciu triedenia prostredníctvom haldy. Bola uprednostnená kvôli tomu, že sa dobre popisuje v rámci objektového modelu a ľahko sa jej porozumie. Tým pádom bolo možné zamerať pozornosť na rozpracovanie zaujímavejších častí algoritmu. Zanechalo to však negatívne následky na dosiahnutom výkone.

Intuitívne sa dá halda predstaviť ako binárny strom vystavaný z indexov ukazujúcich do poľa znakov - reťazca. Každý taký ukazovateľ definuje jediný podreťazec, ktorý sa začína na ním danej pozícií v reťazci. Napríklad, v reťazci S zloženom so znakov $[0..n]$ k indexu $0 \leq i \leq n$ patrí podreťazec znakov $[i..n]$. V minimálnej halde sú ukazovatele zoskupené tak, aby dodržiavali haldovú vlastnosť, čo v tomto prípade znamená: nech je index y uložený v potomkovi uzlu s indexom x, potom podreťazec definovaný indexom x je v

zmysle lexikografického porovnania (tak sú zoradené napríklad prekladové slovníky) menší alebo rovný ako podreťazec definovaný indexom y. Z toho zákonite vyplýva, že index v koreni haldy sa vždy odkazuje na lexikograficky najmenší podreťazec v celom reťazci. Princíp názorne vysvetľuje Obrázok 2.3.



Obrázok 2.3: Princíp minimálnej haldy na triedenie podreťazcov

Vidieť na ňom časť reťazca označeného bielou farbou a časť nad ním zostavenej stromovej štruktúry haldy s prvkami v modrej farbe a minimálnym prvkom zvýrazneným zelenou farbou. V poli sú naznačené prvé štyri znaky a pod nimi čísla reprezentujúce indexy. Rovnaké čísla sa nachádzajú v prvkoch stromu a šípkami ukazujú na prvé znaky podreťazcov, ktoré k nim patria. Možno si všimnúť aj tvar haldy, vždy je naplnená v smere zhora dole a zľava doprava.

Dostupné znaky vytvárajú štyri podreťazce: „abac“, „bac“, „ac“ a „c“. Podreťazec „abac“ s indexom 0 je z nich najmenší, takže sídli na vrchole a spĺňa tak haldovú vlastnosť. Rovnako indexy 2 a 3 v potomkoch vrcholu sú umiestnené správne - podreťazec „ac“ je menší ako jeho jediný potomok, podreťazec „bac“ indexovaný číslom 1, kým uzol patriaci podreťazcu „c“ nemá potomkov žiadnych.

Halda je riešená v module `delheap` a triede `suffix_min_heap` obsahujúcej najmä pole na ukladanie indexov. Pri prechádzaní poľa sa v skutočnosti algoritmus pohybuje ako v strome, využíva fakt, že vďaka danému tvaru haldy sa zo známeho indexu rodiča i odkáže na ľavého potomka indexom $2 * i$ a na pravého potomka indexom $2 * i + 1$. Preťažená metóda `build` zabezpečuje alokáciu pamäte využívanej pre fungovanie haldy a po

konkretizovaní parametrov aj vybudovanie haldy z prvkov udaného poľa, respektíve z jeho udanej časti.

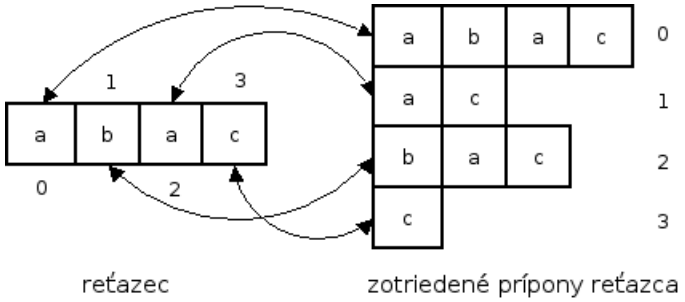
Najprv sa za radom načítavajú všetky indexy z vrchu do pomyselného stromu. V ňom sa rekurzívne testuje či štruktúra spĺňa haldovú vlastnosť, teda žiadané usporiadanie podreťazcov zastúpených indexmi. Ak nie, problematické prvky sa navzájom vymieňajú a overuje sa, či sa ťažkosti nepreniesli do inej časti štruktúry. Za túto operáciu je zodpovedná metóda `heapify` a výsledkom po jej skončení je zotriedená halda. Proces triedenia sprístupňuje funkcia `extract_min`. Odoberá z koreňu haldy najmenší prvok, na jeho miesto dosadzuje posledný prvok stromu a prípadné narušenie, ktoré mohla zmena spôsobiť ošetruje volaním metódy `heapify`.

Azda tou najpodstatnejšou v celej triede je implementácia metódy pre lexikografické porovnanie dvoch reťazcov - `lex_cmp`. Experimenty naznačujú, že práve toto porovnanie zaberá zďaleka najviac času z behu algoritmu a vybraná metóda dramaticky ovplyvňuje ako dlho bude používateľ čakať na výsledok. Zo získaných zistení najlepšie vychádza funkcia `strcmp` zo štandardnej knižnice jazyka C, o zlomok predstihla primitívny `while` cyklus prechádzajúci cez súhlasné znaky poľa. Oveľa pomalšie sa javili funkcie zo štandardnej knižnice STL - `lexicographical_compare` ako aj porovnanie podreťazcov uložených v triede `string` zaostali o niekoľkonásobok. Táto téma je rozpracovaná aj v zhodnotení výsledkov.

2.2 Sufixové pole

Ako už bolo naznačené v úvode, sufixové pole sa hodí na získanie tých podreťazcov nachádzajúcich sa v slovníku, ktoré majú spoločnú predponu. Presnejšie, pre zvolený podreťazec sa požaduje nájdenie iného podreťazca, ak taký existuje, aby bola zhodná predpona čo najdlhšia. Pre tento účel sa používajú dve polia, jednak samotné sufixové pole I a po druhé lexikograficky utriedené pole J.

Pole I vydáva index určujúci podreťazec na ľubovolnom mieste v lexikografickom poradí. Nech $I[i] = j$, potom podreťazec začínajúci sa na pozícií j je i-tym podreťazcom v lexikografickom usporiadaní. Pole J vykonáva opačný prevod, čiže ak $J[j] = i$, i-te miesto v poradí obsadil podreťazec začínajúci sa na pozícií j. Pole J je vlastne inverznou funkciou poľa I, to znamená, že platí vzťah $J[I[i]] = i$ pre všetky i. Aj keď sa jedná o veľmi prostú koncepciu, jej princíp nemusí byť okamžite zrejmý, preto je opäť pripravené grafické zobrazenie súvislostí na Obrázku 2.4.



Obrázok 2.4: Princíp sufixového poľa

Na ľavej strane obrázku je pole štyroch znakov z predošlého príkladu s haldou, na pravej strane sú lexikograficky zoradené všetky jeho podreťazce a vypísané indexy informujúce o ich poradí (celkom vpravo). Prístup do sufixového poľa možno vnímať ako šípku vedúcu z pravej strany na ľavú - zadáva sa doň poradie a získava sa z neho pozícia podreťazca, ktorý toto poradie zaujal. Lexikograficky zotriedené pole potom zastáva úlohu šípky v opačnom smere, treba sa naň obrátiť, keď je vybraná pozícia pre podreťazec a má sa z nej určiť jeho lexikografické poradie. Práve kombinácia týchto dvoch polí poskytuje všetky predpoklady na urýchlenie vyhľadávania zhôd medzi podreťazcami.

Implementácia vyhľadávania pomocou sufixových polí je umiestnená v knižnici `delarray`, trieda sa nazýva `suffix_array`. Podobne ako v prípade haldy ponúka možnosť funkciou `build` alokovať prázdne polia a vyplniť ich obsahom neskôr alebo rovno vytvoriť polia s príslušným zoradením podľa vstupných prvkov.

Prebieha to tým spôsobom, že najprv sa do budúceho lexikograficky usporiadaného poľa vpisujú indexy podreťazcov jednoducho tak ako nasledujú za sebou. Potom pole prevezme halda a vráti ho zotriedené od najmenších podreťazcov po najväčšie. Uvedené poradie sa postupne ukladá a na zodpovedajúce indexy v sufixovom poli sa zaznamenáva pozícia aktuálneho podreťazca. Po zhotovení oboch polí je všetko pripravené na vyhľadávanie.

Funkcia `longest_match` očakáva mimo vytýčenia platných prvkov poľa (prvkov medzi ktorými sa bude vyhľadávať) ako parameter aj index `q` - pozíciu podreťazca ku ktorému sa hľadá zhoda. Pomocou lexikograficky usporiadaného poľa sa zistí jeho poradie, ktoré slúži ako index do sufixového poľa. V ňom sú všetky podreťazce s rovnakou predponou zoskupené v okolí určeného podreťazca, následkom čoho postačuje preskúmať toto okolie.

Najprv sa preberajú predchodcovia, získa sa dĺžka spoločnej predpony prvého z nich a následne každého ďalšieho, až kým sa nenarazí na kratšiu spoločnú predponu alebo začiatok sufixového poľa. Podobne sa spracujú následníci. Do výsledku sa zaznačí dĺžka najdlhšej dosiahnutej predpony a offset smerujúci na pozíciu podreťazca, u ktorého sa objavila. Ak je takých kandidátov viac, vyberá sa ten s najmenším offsetom - vzdialenosťou od pozície zadaného reťazca.

Týmto postupom by sa mala dať identifikovať najdlhšia zhoda v priemere v kratšom čase ako porovnávaním vybraného podreťazca s každým iným podreťazcom v poli, keďže neskúmame tie mimo okolia. Situácia sa dá prirovnať k hľadaniu určitého výrazu v slovníku oproti hľadaniu v nezotriedenom zozname slov. Čo to znamená pre situáciu z Obrázku 2.4?

Dajme tomu, že sa zisťuje najdlhšiu predpona podreťazca „ac“. Porovná sa s „abac“, predošlým podreťazcom v lexikografickom usporiadaní. Zistím sa zhoda v prvom znaku a spolu s offsetom sa zaznačí. Iný predchodca už neexistuje, hľadanie sa teda presunie na následníkov. S podreťazcom „bac“ nenastáva zhoda v prvom znaku, preto sa porovnávanie ukončí. Výsledkom je spätný odkaz s dĺžkou 1 a offsetom 2, keďže v pôvodnom poli je podreťazec „abac“ vzdialený dve pozície od podreťazca „ac“. Namiesto troch porovnaní pre prípad nezotriedených podreťazcov vystačili dve. Pri veľkých dátach sú prirodzene rozdiely omnoho výraznejšie.

Funkcia `longest_match` je preťažená a zároveň s indexom na začiatok a za koniec sufixového poľa sa povoľuje ako parameter zadať tiež index, pod ktorý sa musí zmestiť posledný znak nájdenej zhody ako indikátor konca slovníku. Úprava má svoj pôvod v nepríjemnej chybe, keď rozdelenie na slovník z pôvodného súboru a výhľad z upraveného súboru v ďalšom module nebolo spočiatku korektne premietnuté do metód sufixového poľa. Oprava trvala niekoľko hodín a výsledkom je druhá podoba metódy, tentoraz už rozdeľujúca prehľadávaný priestor.

2.3 Slovník

Na úvod je vhodné upozorniť, že v zdrojových kódoch, komentároch ako aj v tomto texte je v rámci jednoduchosti tam, kde by nemali hroziť nedorozumenia, používané súhrnné pomenovanie slovník pre celú triedu obsahujúcu všetky tu popisované súčasti. Rozlišované sú predovšetkým za účelom presného vysvetlenia ich významu a v tých miestach algoritmu, kde sa s nimi narába oddelene.

Identifikácia a nahrádzanie súhlasných podreťazcov nájdených v slovníku je známa kompresná metóda tvoriaca základ rôznych osvedčených algoritmov. Slovník je dátová štruktúra ktorá sa počas kompresie udržuje a aktualizuje spolu so zmenou pozície v súbore. Nachádza sa v ňom reťazec N predchádzajúcich znakov zo súboru až do aktuálnej pozície. Číslo N reprezentuje veľkosť slovníku. Samotná pozícia v súbore sa dá brať ako index ukazujúci na reťazec od aktuálneho znaku až po koniec súboru. V uvedenej konfigurácií sa pátra po spoločnej predpone oboch reťazcov. Šetrenie miesta v komprimovanom súbore sa dosahuje zakódovaním každej dostatočne dlhej zopakovanej postupností znakov menším záznamom o tom, kde a aká dlhá zhoda sa objavila. Pri obnovovaní pôvodného súboru sa každý opakovaný výskyt podľa uložených inštrukcií jednoducho prekopíruje.

Samozrejme pri rozpoznávaní čo najdlhšej zhody naivným spôsobom hrozí zdĺhavé porovnávanie aktuálneho reťazca so všetkými podreťazcami v slovníku. Ide o nepríjemnosť ktorá sa za desaťročia existencie metódy stala terčom mnohých snáh ako sa niektorým porovnaniam vyhnúť a ako porovnávanie celkovo zefektívniť, či už ušetrením času alebo pamäte. Vyvinuli sa kvalitné techniky, na ktoré sa aj dnes spoliehajú masovo rozšírené nástroje na kompresiu. Za všetky uveďme algoritmus DEFLATE integrovaný v obľúbených programoch zip a gzip.

Po konštrukcií sufixového poľa je pôda pripravená na pomerne rýchle nachádzanie zhôd medzi podreťazcami. Problém je v tom, že vybudovanie sufixového poľa je časovo náročné vzhľadom na potrebu zotriedenia prvkov. Pole sa musí prebudovať pri každom posunutí slovníku, čiže spôsob umiestňovania a posúvania slovníkového okna stojí za zváženie. Postup implementovaný v programe je inšpirovaný prácou Sadakaneho a Imaia [6], ktorí študovali a merali výkon rôznych vylepšení LZ77, medzi nimi aj úpravy s integrovaným sufixovým poľom. Ideou je zavedenie ďalšej štruktúry, a síce výhľadu.

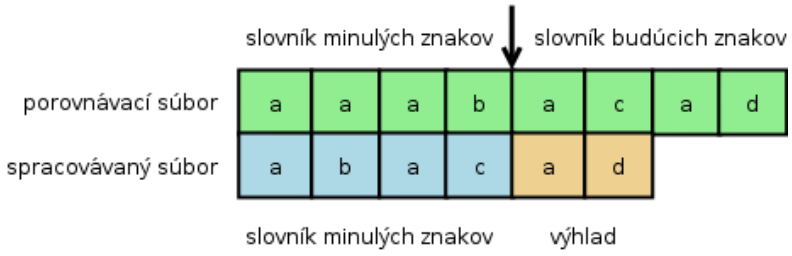
V zásade sa jedná o predĺženie okna slovníku o zlomok či násobok jeho dĺžky. Na začiatku výhľadu sa nachádza znak zo súboru na prvej pozícií za slovníkom, za ním nasledujú ďalšie, až po koniec okna alebo čítaného súboru. Znaky sa spracovávajú ako keby boli v súbore, ale okno sa neposúva. Premiestňuje sa len slovník v rámci okna - sprava doňho pribúdajú znaky z výhľadu. Deje sa tak až do vyčerpania obsahu výhľadu. V tom momente sa celé predĺžené okno posunie o dĺžku výhľadu. Tým pádom sa zníži počet posunutí, po ktorých treba obnoviť sufixové pole a zvýši sa počet podreťazcov, ktoré porovnávame v jednom poli, čím lepšie vynikne výhoda

účinného hľadania.

Riešenie s výhľadom sa pravdaže nedá priamočiaro aplikovať na problém delta kompresie. Zdroj ťažkostí vyplýva zo skutočnosti, že okno je potrebné plniť prvkami z dvoch rôznych súborov. Pri prostom posúvaní vo vnútri okna sa do slovníku dostávajú znaky z výhľadu. Ak bol slovník načítaný z iného súboru ako výhľad, znaky nebudú súhlasiť s pokračovaním slovníkového súboru. V záujme prehľadného východiska z tejto situácie sa vždy udržiava samostatné okno pre pôvodný súbor a samostatné okno pre upravený súbor. V okne pre pôvodný súbor sú slovník a výhľad vytýčené ako dve statické oblasti, ktoré sa nemôžu prelínať, čím sa zabráni problémom s nesúhlasnými znakmi. Slovník v okne pre upravený súbor sa normálne posúva za každým spracovaným znakom z výhľadu.

Ďalšou kľúčovou dilemou je zadefinovať, kde sa vzhľadom k pozícií v upravenom súbore má nachádzať okno v pôvodnom súbore a čo všetko má obsahovať. Pri vývoji sa postupne odskúšalo niekoľko rozmanitých koncepcií sústredených na predchádzajúce prvky a všemožné posuny. Až po čase sa objavila myšlienka, že úpravy zahŕňajú ako pridávanie nových dát, tak aj mazanie dát starých a stálo by za to všímať si taktiež nasledujúce prvky pôvodného súboru a zahrnúť do slovníku eventuálne zhody v tejto oblasti.

Vo výsledku bolo navrhnuté okno pre pôvodný súbor, ktoré obsahuje určitý počet predchádzajúcich prvkov tohto súboru (slovník minulých znakov), rovnaký počet nasledujúcich prvkov tohto súboru (slovník budúcich znakov) a výhľad prevzatý z pôvodného súboru. Okno upraveného súboru vyzerá ako keby upravený súbor neexistoval (slovník minulých znakov a za ním výhľad). Táto schéma už bola uspokojivá z hľadiska dosiahnutého kompresného pomerom. Pri bližšom objasnení pomôže Obrázok 2.5.



Obrázok 2.5: Podoba a rozmiestnenie slovníkov a výhľadu

Horné pole patrí pôvodnému súboru, dolné spracovávanému upravenému

súboru. Šípka ukazuje na aktuálnu pozíciu, až na okrajové prípady rovnakú pre oba súbory. V takomto rozostavení okno pre pôvodný súbor zahŕňa dva slovníky, obidva s pozadím v zelenej farbe a k nim sprava pripojený výhľad z prvkov upraveného slovníku s pozadím v oranžovej farbe. Čo sa týka okna v upravenom súbore, začína sa modro zafarbeným slovníkom doplneným oranžovom zafarbeným výhľadom. Výhľad v jednom aj druhom okne je teda ten istý, podobne ako pozícia.

Keď sa minú načítané znaky, pozícia v súboroch sa posunie o dĺžku výhľadu. Popri intuitívnom postupe presúvania sa ešte zvažovalo dočasné vyčkávanie so zmenou pozície v pôvodnom súbore po dlhej zmene. Tým sa mal vystihnúť stav, pri ktorom bolo do pôvodného súboru vložené menšie množstvo dát. Ak by sa pozdržalo posunutie pozície kým sa tieto dáta prekonajú, dalo by sa plynulo nadviazať na prerušenú zhodu, navyše by sa ušetrila zbytočná rekonštrukcia sufixového poľa z pôvodného súboru. Lenže zmysluplnosť tohto nápadu, predovšetkým dopad na kompresný pomer v nesúvisiacich súboroch, by sa musela overovať zložitým experimentovaním. Preto zostalo len pri základnom posune.

Počas pohybu v súboroch môže nastať niekoľko hraničných prípadov. Pri vysporiadavaní sa s nimi si zväčša stačí uvedomiť, že upravený súbor je ten, pre ktorý sa konštruuje delta súbor a dbať na to, aby bol v ľubovolnej pozícií výstup v tomto zmysle korektný. Hneď na prvé úskalie sa narazí na počiatočnej pozícií. Nie sú k dispozícií žiadne znaky, ktorými by sa dali naplniť slovníky smerované do minulých dát. Pravidlo je jednoduché: ak nie je dostatok znakov na vyplnenie niektorého zo slovníkov, jeho obsah sa zahodí a celý prepíše nulovými znakmi. Na nultej pozícií v súbore sa teda pripraví jedine slovník pre znaky nasledujúce v pôvodnom súbore (pravý zelený slovník), zvyšné slovníky sa ponechajú prázdne.

Na druhej strane hrozí, že pri načítavaní prvkov do okna sa objaví koniec niektorého súboru. Znovu sa budeme riadiť jasnou poučkou: v ďalšej činnosti sa pokračuje len ak sa podarilo načítať do výhľadu aspoň jeden znak zo zmeneného súboru. Pre pôvodný súbor platí dodatok: ak zo súčasnej pozície nie je program schopný naplniť aspoň jeden slovník, ďalej sa už nepokúša vkladať do okna výhľad, považuje pôvodný súbor uzavretý a ďalej ho neposúva.

Po vyčerpávajúcom zoznámení s navrhnutou modifikáciou slovníkovej metódy bude predstavený zastrešujúci súbor `deldict` a všetky techniky vyhľadávania dostupné v triede `delta_dictionary`. V prvom rade je nevyhnutné naviazať okná na dva porovnávané súbory metódou `assoc_files`. Rovnako je nutné vyhradenie pamäte, nastavenie okien na východziu pozíciu

a zostavenie sufixových polí, čo má na starosti funkcia `set_dict`. Tým sa ukončí príprava nástrojov na vyhľadávanie. V tej chvíli stačí už iba zadať pozíciu podreťazca z výhľadu, pre ktorý sa požaduje nájdenie náhrady a vybrať si, ktorý z troch postupov sa použije.

Prvým je `dictionary_search`, základné hľadanie v rámci slovníku. Dĺžka spoločnej predpony je v ňom limitovaná pravým okrajom slovníku (pôvodný súbor) respektíve pravým okrajom okna (upravený súbor). Uvažujúc o situácií na Obrázku 2.5: vyberie sa prvý podreťazec z výhľadu, „ad“, a zistí sa, či má s nejakým podreťazcom zo slovníku spoločnú predponu. Slovník z upraveného súboru dáva najlepšiu zhodu dĺžky 1 a offsetu 2 (podreťazec „acad“), slovník z pôvodného súboru vráti výskyt s rovnakým offsetom (výhľad sa k slovníku pripája sprava) a dĺžkou 2 (podreťazec „ad“). Podobne ako pri porovnávaní v sufixovom poli sa dá prednosť čo najdlhšiemu a najbližšiemu výsledku. Keby sa žiadna rovnaká predpona nenašla, znamenalo by to, že podreťazec na súčasnej pozícií sa nedá ničím nahradiť. Jeho prvý znak by sa ponechal tak ako je a vyskúšal by sa podreťazec začínajúci jeho druhým znakom. V predvedenej demonštrácií sa ale môže podreťazec „ad“ prepísať inštrukciou na skopírovanie dvoch znakov vzdialených dve pozície smerom vzad. Čiže dva znaky sa preskočia aj vo výhľade. Tým sa dosiahne jeho koniec.

Pre predstavu, nech upravený súbor vznikol z pôvodného vymazaním prvých dvoch znakov a oba sú dlhšie ako viditeľná časť. To znamená, že pôvodný súbor pokračuje za slovníkom rovnakými znakmi ako upravený súbor za výhľadom a získaná spoločná predpona sa dá rozšíriť. Základné hľadanie však o tom neinformuje, keďže ostatné znaky presahujú slovník. Na predĺženie zhody slúži `extended_search`, rozšírený variant hľadania. Ak sa počas overovania dĺžky spoločnej predpony narazí na koniec slovníku (pôvodný súbor), prípadne okna (upravený súbor), namiesto toho, aby sa hľadanie zastavilo, načíta sa ďalšia časť súboru a skúsia sa porovnať znaky na začiatku. Celý proces sa opakuje až kým sa neobjaví rozdiel, potom sa vrátia súbory do pôvodného stavu. Niektoré predĺženia budú samozrejme neúspešné, nejde však o priveľmi náročnú operáciu a najmä v podobných súboroch časté predĺženia ušetria množstvo prostriedkov na zbytočné základné hľadanie v predĺžení zhody. Preto sa `extended_search` vyplatí používať, čo sa ukázalo aj vo výsledkoch testov v tretej kapitole.

Posledným stupňom je takzvané lenivé hľadanie alebo `lazy_search`. Postupuje presne tak isto ako rozšírené hľadanie, no ak zistí rovnakú predponu, nepošle ju ihneď na výstup, ale ju uloží do interných štruktúr. Pri novej

požiadavke na nájdenie náhrady porovná výsledok s uloženou zhodou. Ak je nová zhoda dlhšia, stará sa skráti na prvý znak predošlého podreťazca. V opačnej situácií sa nový výskyt ignoruje a za nájdený je prehlásený ten starý. Po rozhodnutí nastaví funkcia pozície v oboch súboroch tak, aby zodpovedali výstupu. Inak povedané, lenivé hľadanie zabráni tomu, aby sa prijatím kratšej zhody prišlo o dlhšiu zhodu hneď za ňou.

Po dosiahnutí pravého okraja výhľadu čaká na okná presunutie o dĺžku výhľadu a rekonštrukcia ich polí. To implementuje metóda `shift_dict`. Odohrali sa rôzne pokusy o čo najlepšiu optimalizáciu tejto funkcie, keďže bola podozrivá zo spomaľovania behu programu. Keď to posun umožňuje, využije sa čo najviac dát z minulej podoby štruktúry a minimalizuje sa počet diskových operácií. Ukázalo sa však, že napríklad čítanie väčšieho množstva dát z binárneho súboru metódou `read` spotrebováva len nepatrný čas oproti stavbe sufixového poľa, dokonca aj v neveľkých slovníkoch. To zmarilo akékoľvek snahy o iné vylepšenia rovnakého charakteru, čo sa prejavilo na veľmi priamočiarom návrhu kódovača a dekódovača. Klasické posúvanie dopĺňa ešte metóda pre lenivé posúvanie, `lazy_shift`, ktorá ošetruje pozíciu v súboroch pri vracaní sa k predchádzajúcim zhodám.

2.4 Prevodník

Výsledok požiadavky na nájdenie prechádzajúceho výskytu podreťazca v slovníku môže byť dvojaký. Buď sa žiadny výskyt nenájde a slovník oznámi znak čo sa má zaznamenať nezmenený alebo výskyt nastane a slovník zostaví kopírovaciu inštrukciu. Pre slovník s lenivým vyhľadávaním sa navyše povoľuje oznámenie, že čaká na výsledok ďalšej požiadavky. Tieto informácie treba spracovať a vhodne ich reprezentovať za účelom uloženia v delta súbore. Obyčajný znak z binárneho súboru môže mať akúkoľvek hodnotu a inštrukcie sa majú zapísať tak, aby sa dali rozumne odlíšiť. Predpoklady na úspešné zvládnutie spomenutých problémov poskytuje prevodník zastupujúci úlohu prostredníka medzi slovníkom a kódovačom.

V knižnici `delsymbol` je nadefinovaná vlastná malá abeceda so štyrmi druhmi symbolov. Sú v nej prvky (znaky), ktoré sa nebudú nahrádzať (typ `literal`) a ich udaná hodnota, ďalej kopírovacie inštrukcie (typ `match`) s dĺžkou, offsetom a príznakom či sa má kopírovať z pôvodného alebo upraveného súboru. Predposledný je zvláštny symbol indikujúci, že výsledok lenivého hľadania nie je úplne doriešený (typ `pending`) a zostáva neplatný symbol (typ `invalid`) pre osobitné situácie ako je vyvolanie výnimky alebo

dosiahnutie konca súboru, kedy sa nedá uplatniť žiadny z ostatných symbolov. Prevodník získava dáta zo slovníku, podľa ich charakteru vytvorí symbol príslušného typu a dá ho k dispozícii kódovaču.

Popri tom prevodník ešte riadi a vytvára rozhranie pre vyhľadávanie v slovníku. Na podnet kódovača dáva pokyn na načítanie súborov a inicializáciu polí. Vždy keď je potrebné zakódovať symbol, požiada o hľadanie v slovníku. Obdržané výsledky premieta do aktuálnej pozície, čím vlastne vykonáva pohyb v slovníku. Tiež iniciuje posunutie slovníku vo chvíli vyčerpania všetkých znakov. Disponuje priestorom určeným na uschovanie nedoriešených zhôd pochádzajúcich z lenivého vyhľadávania. Pre každý typ hľadania v slovníku má definovanú samostatnú riadiacu funkciu, jednotnou metódou `element_to_symbol` sa dá pohodlne pristupovať ku každej z nich zvolením parametru `mode`.

2.5 Kódovač

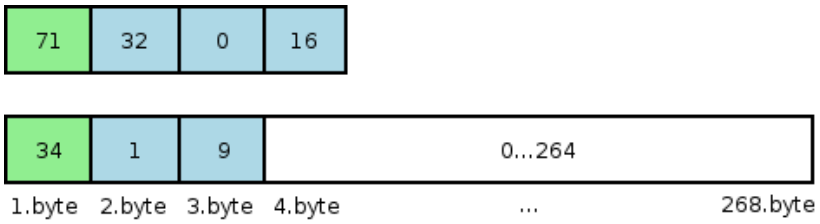
Posledný krok kompresie sa sústredí na vytvorenie samotného delta súboru. Jeho výsledná veľkosť priamo závisí na forme zvolenej pre ukladanie dát. Dôležitý je výber prostriedkov, ktorými sa dosiahne rozlíšenie rôznych druhov symbolov. Do úvahy prichádza povedzme účinné kódovanie spojené s úspornou definíciou abecedy symbolov. Na princípe tohto typu stavia s úspechom už spomínaný DEFLATE. Súčasťou projektu bol taktiež menší prieskum týkajúci sa aritmetického kódovania, bohužiaľ primitívne modely neprimerane brzdili chod algoritmu a náročnejšie z nich sa nepodarilo riadne naimplementovať. Objektový model riešenia je však pripravený na eventuálne zakomponovanie pokročilejšieho kódovania bez zbytočných prekážok.

Každopádne, v súčasnej implementácií sa uplatnil blokový formát súboru v module `delblock`. Bloky majú premenlivú dĺžku a v každom z nich smie byť uložený iba jeden typ symbolu. V skutočnosti stačí uchovávať dva z typov, prvky a kopírovacie inštrukcie. Typy sa spoznávajú podľa prvého bytu hlavičky, ktorou každý blok začína. Prvý byte taktiež určuje v koľkých nasledujúcich bytoch sa ešte hlavička nachádza. Aj hlavičky totiž môžu mať rozdielnu veľkosť.

Presnejšie, v bloku obsadenom prvkami má hlavička okrem prvého bytu povolený ďalší 1 až 4 byty na uschovanie informácie koľko pôvodných prvkov je vlastne za hlavičkou uložených. Na ukladaný údaj sa samozrejme použije len toľko bytov, koľko je nevyhnutne potrebných, preto premenlivá dĺžka. Ak je na vstupe väčší počet prvkov ako číslo, ktoré sa zmestí do 4 bytov,

očakáva sa, že zvyšné prvky sa uložia v novom bloku s čistou hlavičkou.

Čo sa týka predchádzajúcich výskytov reťazcov, v hlavičke tohto bloku je uložená priamo kopírovacia inštrukcia. Údaje o dĺžke a offsete môžu zaberať od 1 bytu do maximálne 4 bytov každý. Na odlíšenie príznaku z ktorého súboru inštrukcia pochádza sa využije zvyšné voľné miesto v prvom byte hlavičky. Tým pádom, aby sa oplatilo uložiť krátku kopírovaciu inštrukciu, musí mať dĺžku aspoň 3 byty (hlavička + dĺžka + offset), kratšie inštrukcie sa zahadzujú. Hlavičky sa binárne čítajú a zapisujú po jednotlivých bytoch. Ukážku štruktúry bloku vidieť na obrázku 2.6.



Obrázok 2.6: Štruktúra bloku pre inštrukciu a pre prvky

Prvý byte hlavičky sa spozná podľa zelenej farby. V prvom prípade ide o blok s kopírovacou inštrukciou namierenou do pôvodného súboru s hodnotou dĺžky do 1 bytu a hodnotou offsetu do 2 bytov. V zostávajúcich modrých bunkách hlavičky nasledujú uložené dáta, po započítaní bitových posunov odhalíme zhodu dĺžky 32 s offsetom 4096. Dolný blok obsahuje prvky v počte, na ktorého záznam sa spotrebovali 2 byty. Z modrej časti hlavičky sa dozvedáme, že posledný platný index pri počítaní od nuly je 264. Ukážkové hodnoty celkom nezodpovedajú realite. Je to spôsobené tým, že štandardné funkcie na čítanie a zapisovanie do binárnych súborov pracujú so znamienkovými znakmi a aby sa zabránilo deformáciám pri nežiadúcom pretypovaní zo šablón, hodnoty hlavičky sa pred zapísaním centrujú. Špeciálne prázdne typy bloku sú vyhradené pre nedoriešené bloky v prípade pretečenia pri dekódovaní (viz. nasledujúca podkapitola) a pre neplatné symboly.

Pred začatím kompresie dát zo súboru sa zapíše na výstup inicializačný blok s dĺžkou slovníku. Pri dekompresií sa tento údaj využije na prepočítanie offsetov z pôvodného súboru voči pozícií aktuálneho súboru. Kódovanie funguje tým spôsobom, že sa prostredníctvom prevodníka prijímajú symboly konvertované zo znakov a inštrukcií vyprodukovaných slovníkom. Rozdeľujú sa na nedoriešený symbol, symbol prvku a symbol kopírovacej inštrukcie a

každý sa ošetruje zvlášť.

Keď je na vstupe nedoriešený symbol, nerobí sa nič, okrem toho že sa vyžiada ďalší symbol, aby sa mohlo pokračovať. Symbolu prvku sa ujme funkcia `literal_to_block`, na začiatku pripravovaného bloku sa vyhradí maximálne miesto aké môže hlavička zaberať a za ním sa blok plní prvkami až kým sa nevyčerpá kapacita bloku alebo sa na vstupe neobjaví iný typ symbolu. Vtedy sa metóda vráti do hlavičky a za účasti bitových posunov v nej uvedie typ a konečnú reálnu veľkosť bloku. Posledný nezapísaný symbol prenechá na ďalšie spracovanie. Konečne z prijatej kopírovacej inštrukcie metóda `match_to_block` dešifruje presný typ bloku, ktorý použije a dĺžku jeho hlavičky. Potom do nej zapíše vstupné hodnoty. Načíta nový symbol a odovzdá ho kódovaču. Prevodník po vymíňaní všetkých znakov odošle kódovaču neplatný symbol ako indikátor konca súboru (či chyby).

2.6 Dekódovač

K dotvoreniu funkčného návrhu na riešene problému delta kompresie chýba posledný, ale mimoriadne dôležitý prvok a to trieda, ktorá bude schopná aplikovať zmeny popísané v delta súbore na pôvodný porovnávací súbor a obnoviť upravený súbor, z ktorého boli rozdiely generované, v nezmenenej podobe. Služby s tým súvisiace sú implementované v súbore `deldecoder`. Činnosti vykonávané pri dekompresií pripomínajú povinnosti vykonávané v kompresií kódovačom. Z delta súboru sa načítajú úvodné inicializačné byty. Hodnota uschovaných offsetov z pôvodného súboru je totiž závislá na dĺžke slovníka, ktorý našiel príslušné inštrukcie a bez nej by nebolo možné správne prepočítať dotknuté indexy. Ďalej sa začína každý krok rovnako, preskúma sa prvý byte hlavičky, čím výjde na javo celková dĺžka hlavičky a to, či za ňou nasledujú prvky alebo treba vykonať kopírovaciu inštrukciu.

Obnovenie prvkov metódou `literal_block_to_elements` je úplne triviálne. Podľa hlavičky sa vytýči hranica kam siahajú prvky a pripravia sa na výstup. Nakoniec sa dá pokyn na načítanie typu ďalšieho bloku. Vykonanie kopírovacej inštrukcie tiež nie je zvlášť zložité, ale číha v ňom niekoľko nebezpečenstiev. Po prvé si treba premyslieť ako správne vypočítať indexy a nastaviť pozíciu (predovšetkým v pôvodnom súbore), z ktorej sa majú kopírovať prvky. Po druhé, príliš dlhá zhoda pretečie cez pole používané na prevod bloku alebo na konci súboru môže byť inštrukcia pretekajúca cez slovník. To zapríčiňuje použité rozšírené vyhľadávanie, ktoré dovoľuje pri presiahnutí okna opakovať prvky zo slovníku, aj keď je už výhľad prázdny,

aby sa dosiahla čo najlepšia zhoda.

Funkcia `match_block_to_elements` teda po zistení dĺžky a offsetu bloku skontroluje hrozbu pretečenia. Keď má k nemu dôjsť, súbory sa nastavia na potrebnú pozíciu, ale skopíruje sa iba toľko znakov, aby nepretiekli. Spracúvaná zhoda sa skráti o práve vyriešené znaky a patrične sa upraví tiež offset zhody. Keďže sa vyskytlo pretečenie, nastaví sa `pending_block` ako typ bloku, aby sa vedelo, že sa má v ďalšej fáze ošetriť. Potom sa tento proces opakuje vo funkcií `pending_block_to_elements`, bez prvotného nastavovania pozície v súbore, až kým zo vstupu neubudne toľko znakov, aby sa zmestili do poľa. Tie sa už normálne prekopírujú na výstup a vyzistí sa, čo je uložené v nasledujúcom bloku.

Kapitola 3

Vyhodnotenie riešenia

Prvé spustenie algoritmu na reálnych dátach vyvolalo mierne sklamanie a rozpaky. Kým na malých príkladoch si program počínal celkom dobre a správal sa podľa očakávaní aj v hraničných situáciách, na súboroch veľkosťou presahujúcich pár málo desiatok kilobytov sa čakanie na výsledok vyšplhalo na niekoľko sekúnd. Po tomto zistení bolo vynaložené nemalé úsilie a vďaka nemu sa podarilo beh mierne zrýchliť a identifikovať príčiny slabšieho výkonu oproti ostatným konkurentom. Pred tým ako budú v stručnosti predstavené uvedené zistenia, je vhodné vysvetliť ako sa program používa.

3.1 Použitie programu

Program používateľovi ponúka ovládanie prostredníctvom príkazového riadku. Očakávaný príkaz ma tento tvar:

```
adelta [-adm] encode orig-file rev-file delta-file
adelta [-adm] decode orig-file delta-file rev-file
```

Parametre a voľby majú význam vyznačený v Tabuľke 3.1. Vždy sa predpokladá použitie aspoň jedného z príkazov `encode` alebo `decode` doplneného cestami k príslušným súborom: `orig-file` je názov súboru voči ktorému sa porovnáva, `rev-file` je názov súboru ktorý sa porovnáva a `delta-file` je rozdielový súbor. Treba dbať na správne poradie týchto súborov pri zadávaní požadovanej operácie.

Parameter	Význam	Typ	Rozsah
`-a`	Dĺžka výhľadu (pomer)	double	(0.0,1.0]
`-d`	Dĺžka slovníku	int	[5,`maxint`]
`-m`	Spôsob vyhľadávania	int	{1,2,3}
`encode`	Vytvorenie delta súboru	char *	-
`decode`	Obnovenie z delta súboru	char *	-
`orig-file`	Pôvodný súbor	char *	-
`rev-file`	Upravený súbor	char *	-
`delta-file`	Delta súbor	char *	-

Tabuľka 3.1: Význam, typ a rozsah vstupných parametrov

Ak sa používateľ chystá uviesť niektorú z volieb `-a`, `-d` alebo `-m`, musí ju zadať pred určením výkonného príkazu, inak bude nastavenie ignorované. Dĺžka výhľadu je desatinné číslo zadávané v tvare $x.yz$. Musí byť väčšie ako 0 a nesmie presahovať hodnotu 1. Ide vlastne o pomer k dĺžke slovníka, ak `-a` $= 1$, potom je výhľad rovnako dlhý ako slovník. Dĺžka slovníka musí byť z implementačných dôvodov aspoň 5 znakov. Teoreticky je možné využívať slovníky dĺžky až do maximálnej hodnoty typu `integer`. Spôsob vyhľadávania sa určuje nasledovne: 1 - základné vyhľadávanie, 2 - rozšírené vyhľadávanie, 3 - lenivé vyhľadávanie. Nešpecifikované voľby sa nahradia implicitnými hodnotami.

Nastavovaním dĺžky slovníku a výhľadu a spôsobu vyhľadávania možno upravovať kompresný pomer a rýchlosť kompresie ak sú implicitné hodnoty nevyhovujúce. Na vyhľadávanie sa súčasne používajú tri slovníky, celá štruktúra bude mať teda trojnásobok zadanej dĺžky. Program dosahuje problematický výkon v zmysle uplynutého času na textových súboroch väčších ako 50 KB a na binárnych súboroch celkovo. Program sa dá využiť napríklad pri správe softvérových projektov rozdelených do menších modulov na generovanie delta súborov pri ukladaní revízií. Vyslovene sa nehodí na cielenú kompresiu väčších súborov.

3.2 Dosiahnuté výsledky

Pri obmedzených podmienkach a postupoch, ktorými sa testoval výkon programu sa nedajú dosiahnuť reprezentatívne výsledky a výkon použitej kompresnej metódy je ovplyvnený nedokonalým návrhom programu. Na druhej

strane, výsledné údaje nie sú úplne nezaujímavé, čo je dôvod prečo sa pre ne nakoniec našlo miesto v tejto práci.

Riešenie sa testovalo v laboratóriu fakulty na počítači s procesorom Intel Pentium 4 taktovanom na 2,66 GHz, 1,00 GB vnútornej pamäte a operačným systémom Microsoft Windows XP. Kód bol preložený v rámci vývojového prostredia Microsoft Visual Studio 2005 so štandardnými nastaveniami. Ako testovacie dáta bolo vzhľadom na možnosti zvolených desať náhodne vybraných súborov z projektu 7Zip nachádzajúceho sa na stránkach SourceForge. Ide o súbory z revízie 4.20 a ich náprotivky z revízie 4.52, ktoré boli doplnené zálohou z vývojovej verzie tohto programu. Celková veľkosť súborov bola približne 500 KB a všetky sú uložené v samostatnej zložke zaradenej do projektu. Analyzovala sa časová zložitosť, spoliehajúc sa na priemerné výsledky získané z knižníc `time.h` a `assert.h` jazyka C, a dosiahnutý kompresný pomer. Prvý test bol zameraný na vplyv dĺžky slovníka, zaznamenané dáta sú uvedené v Tabuľke 3.2.

	1KB	4KB	8KB	16KB	32KB
Čas	1.32	1.50	1.33	0.97	0.59
Kompresia	0.30	0.20	0.23	0.39	0.78

Tabuľka 3.2: Vplyv dĺžky slovníka na výkon

V záhlaví sú zoradené testované hodnoty pre dĺžky slovníka s rozumným dosiahnutým kompresným pomerom. Každý z nich mal nastavenú dĺžku výhľadu na 0.5 a lenivý spôsob hľadania. Položka čas udáva priemerný počet sekúnd potrebných na spracovanie jedného súboru, kým záznam kompresia informuje o priemernej dĺžke komprimovaného súboru vzhľadom na nekomprimovaný súbor. Najvýhodnejšie v pomere času a ušetreného priestoru sa pre testované dáta javia slovníky veľkosti 4KB a 8KB. Väčšie slovníky trpia tým, že priemerná veľkosť jedného súboru je menšia ako 23KB, čiže napríklad 32KB slovník sa ani nemá šancu naplniť (preto je jeho čas taký malý). V ďalších nastaveniach sa tip na dĺžku potvrdil, zvyšné úvahy boli rozvíjané pre dva najúspešnejšie varianty. Dopad dĺžky výhľadu bol pre ne veľmi podobný, úplne zlyhali malé zlomky, kým nad hodnotou 0.5 to bol vždy kompromis medzi čoraz vyššou rýchlosťou a čoraz horším kompresným pomerom. Ukážku hodnôt reprezentuje 4KB slovník s lenivým hľadaním, jeho výsledky sú v Tabuľke 3.3 pre dĺžky výhľadu od 0.25 do 1.0.

Zostáva overiť ako je to so spôsobom vyhľadávania, vyskúšané boli obe

	0.25	0.5	0.75	1.0
Čas	2.54	1.50	1.18	0.97
Kompresia	0.21	0.20	0.22	0.24

Tabuľka 3.3: Vplyv dĺžky výhľadu na výkon

konfigurácie. Opäť u oboch slovníkov prepadlo jednoduché vyhľadávanie. Určiť víťaza medzi rozšíreným a lenivým hľadaním je ťažké, keďže výsledky sú veľmi tesné a v niektorých prípadoch sa na prvej pozícií tieto spôsoby striedajú. Pre predstavu, údaje zaznamenané pre 4KB slovník s dĺžkou výhľadu 0.5 sú zapísané do Tabuľky 3.4 podľa rôznych spôsobov.

	Základné	Rozšírené	Lenivé
Čas	1.62	1.48	1.50
Kompresia	0.21	0.20	0.20

Tabuľka 3.4: Vplyv spôsobu vyhľadávania na výkon

Testy naznačujú, ktorými slovníkmi sa najefektívnejšie zkomprimuje testovaný súbor dát a ako treba upraviť dĺžky, aby sa dosiahla buď vyššia rýchlosť alebo účinnejšia redukcia veľkosti. Tieto poznatky boli využité pri zisťovaní, ktorá časť algoritmu je časovo najnáročnejšia. Pomerne dlho sa v návrhu pracovalo na minimalizácií operácií so súbormi. Keď to neviedlo k zlepšeniu, pozornosť sa upriamila na hľadanie v sufixovom poli. Rôznymi opatreniami na obmedzenie prehľadávania za cenu plytvania priestorom na disku sa zisťoval časový podiel hľadania, ale nikdy sa nepodarilo zrýchliť algoritmus o viac ako jednotky percent. Pri kontrole ostatných tried sa v halde overoval aj `while` cyklus, ktorý zabezpečoval lexikografické porovnanie. Nahrádzal sa rôznymi vstavanými funkciami a pozoroval sa dopad na beh algoritmu programu. So staršou funkciou `strcmp` postup fungoval, no pri prechode do STL nastali problémy s rýchlosťou. Kompresia na testovacích dátach bola spustená so štyrmi rôznymi funkciami pre porovnanie: naivným `while` cyklom, funkciou `strcmp`, funkciou `lexicographical_compare` (označená ako `lex_cmp`) a funkciou `compare` triedy `string`, do ktorej sa uložili porovnávané reťazce. Použil sa 4KB slovník so štandardným nastavením, teda veľkosťou výhľadu 0.5 a lenivým vyhľadávaním. Ako pokus dopadol popisuje tabuľka 3.5.

Funkcia `strcmp` je silne optimalizovaná pre operácie nad reťazcami zna-

	`while`	`strcmp`	`lex_cmp`	`compare`
Čas	1.53	1.50	6.97	43.6
Kompresia	0.20	0.20	0.20	0.20

Tabuľka 3.5: Porovnanie rôznych funkcií pre lexikografické porovnanie

kov, na vysokých časoch STL sa však mohli podpísať aj špecifiká implementovaného návrhu. Tak či tak, nakoniec sa v ňom pri porovnávaní reťazcov uplatnila práve knižničná funkcia jazyka C.

Na záver je prichystané vyhodnotenie ako projekt obstál v porovnaní klasickým postupom ako sa vysporiadať s tvorbou rozdielov, kombináciou programov diff a gzip. Ak by sa bral do úvahy čisto diff, delta kompresia jednoznačne víťazí vo veľkosti výstupných súborov. Kým diff súboru dát dosahuje veľkosť 227KB, delta súbory obsadzujú iba 52KB. Nanešťastie je to za cenu veľkej nevýhody v rýchlosti. Kým kompresií trvá spracovať jeden súbor v priemere vyše sekundy, najväčší diff zo súboru sa vytvorí za jednu desatinu sekundy. Do ešte menej šťastnej pozície ju stavia následná kompresia v gzip-e. Tá zvýši celkový čas na niečo vyše troch desatín sekundy pre najhorší prípad a zmenší celkovú veľkosť pod objem zabraný delta súbormi. Špecializované nástroje na delta kompresiu zväčša vylepšujú výsledky diff-u a gzip-u, dá sa teda s veľkou pravdepodobnosťou predpokladať, že projekt by príliš dobre neobstál ani v porovnaní s ostatným programami. Potenciálnou silnou stránkou riešenia by mohla byť priestorová zložitosť vo vzťahu k operačnej pamäti, tú sa však nepodarilo otestovať. Zradnou vlastnosťou algoritmu so sufixovým poľom je nevyrovnaný výkon. Aj pri normálnom používaní nástroja sa dá postrehnúť, že doba kompresie nezávisí len od veľkosti spracovávaných dát, ale aj ich charakteru. Na väčšinu binárnych súborov je potom nepoužiteľný, keďže aj pre menšie podobné PDF súbory sa delta súbor vytvára niekoľko minút.

3.3 Diskusia o vylepšeniach

Predstavená implementácia delta kompresie s využitím sufixových polí trpí množstvom nedostatkov, pre ktoré existujú riešenia. Ak by sa vylepšila, pozície v porovnaní s klasickými metódami by sa mohli veľmi rýchlo vymeniť. Hlavným kandidátom na prepracovanie je určite proces konštrukcie sufixového poľa. Predtriedenie prvkov je dnes dávno prekonané špecializovanými

algoritmami na priame vybudovanie poľa za podstatne kratší, aj lineárny, čas bez veľkej spotreby pamäte ako predviedli v svojom článku Ko a Aluru [7]. Táto operácia je (ako ukázal aj profil) pritom najnáročnejšia z celého programu, dopad zrýchlenia by bol teda veľmi výrazný. Bolo by tiež možné napraviť množstvo chýb týkajúcich sa spôsobov triedenia prvkov (efektívnejší algoritmus) či vyhľadávania opakovaných výskytov reťazcov (zavedenie limitov pre dobu provnávania). Ďalšia časť, ktorá by si zaslúžila viac pozornosti je algoritmus na posúvanie slovníkového okna. Stratégia uplatňovaná na výbery prvkov zo súboru rozhoduje nielen o počte zhôd, ktoré sa dosiahnu pri vyhľadávaní spoločnej predpony podreťazcov, ale aj o počte posunutí slovníku. Tým by sa mohol získať ďalší čas k dobru.

Čo sa týka veľkosti delta súborov, súčasný variant kódovania a formátu súboru je len náhradou za dokonalejšie riešenie, na ktoré sa myslelo od začiatku projektu. Moffat, Neal a Whitten totiž publikovali výborný článok a k nemu niekoľko dodatkov, v ktorých prezentujú nástroje na zrýchlenie aritmetického kódovania a vybudovanie šikovných modelov nevyhnutných na jeho efektívne fungovanie [8]. Je škoda, že sa mi nepodarilo niektorý z ich nápadov zakomponovať do mojej implementácie a určite by stálo za to sa k tejto myšlienke niekedy vrátiť a preskúmať ako by tieto dva princípy dokázali spolu fungovať.

Kapitola 4

Záver

Tvorba tejto bakalárskej práce bola pre mňa mimoriadne vyčerpávajúcou a zároveň užitočnou skúsenosťou. Zo začiatku som nemal príliš jasnú predstavu aké problémy chcem riešiť a ako by mal môj projekt vlastne vyzerať. No s pribúdajúcimi načítanými článkami a časom stráveným za počítačom úspešnými aj neúspešnými pokusmi začalo pribúdať toľko nových nápadov, že som mal zrazu opačný problém - čomu sa venovať skôr. To nakoniec spôsobilo, že nie všetky plány, ktoré som si vytýčil sa mi podarilo aj naozaj splniť. Napriek tomu si myslím, že práca na tomto projekte bola veľkým prínosom. Naučil som sa ako získavať potrebné informácie, ako navrhovať vlastné riešenia, ako ich triezvo posúdiť a v neposlednom rade tiež ako o tom napísať pre ostatných tak, aby si aspoň časť z tohto prínosu odniesli tiež. Výsledok, samozrejme, nie je dokonalý. Myslím si však, že tie najdôležitejšie ciele sa mi splniť podarilo. Preskúmal som problematiku použitia sufixových polí a jej možností rovnako ako slabín v kontexte delta kompresie. Naimplementoval som vlastný algoritmus s riešením mnohých zaujímavých problémov. Ponúkol som jeho kritiku a zároveň nápady ako opraviť nedostatky ktorých som sa dopustil. Zostáva mi len dúfať, že pre čitateľa bolo stretnutie s touto prácou rovnakou príjemným zážitkom ako pre mňa.

Literatúra

[1] Free Software Foundation: *Comparing and Merging Files*, http://www.gnu.org/software/diffutils/manual/diff.html, 2002.

[2] Korn D. G., Vo K. P.: *Vdelta: Differencing and Compression*, Practical Reusable Unix Software, John Wiley and Sons, 1995.

[3] Ziv J., Lempel A.: *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory **23** (1977) 337-343.

[4] Hunt J. J., Tichy W. F., Vo K. P. *Delta Algorithms: An Empirical Analysis*, ACM Transactions on Software Engineering and Methodology **2** (1998) 192-214.

[5] Manber U., Myers G.: *Suffix arrays: a new method for on-line string searches*, SIAM Journal on Computing **5** (1993) 935-948

[6] Sadakane K., Imai H.: *Improving the Speed of LZ77 Compression by Hashing and Suffix Sorting*, IEICE Trans. Fundamentals **12** (2000) 2689-2698

[7] Ko P., Aluru S.: *Space efficient linear time construction of suffix arrays*, Journal of Discrete Algorithms **5** (2005) 143-156

[8] Moffat A., Neal R. M., Whitten I. H.: *Arithmetic coding revisited*, ACM Transaction on Information Systems **16** (1998) 256-294